Learning Outcomes

- An understanding of the structure and limits of multiprocessor hardware.
- An appreciation of approaches to operating system support for multiprocessor machines.
- An understanding of issues surrounding and approaches to construction of multiprocessor synchronisation primitives.

Multiprocessor Systems

Chapter 8, 8.1

Multiprocessor System

- We will look at shared-memory multiprocessors
  - More than one processor sharing the same memory
- A single CPU can only go so fast
  - Use more than one CPU to improve performance
  - Assumes
    - Workload can be parallelised
    - Workload is not I/O-bound or memory-bound
- Disks and other hardware can be expensive
  - Can share hardware between CPUs

Amdahl’s law

- Given a proportion $P$ of a program that can be made parallel, and the remaining serial portion $(1-P)$, speedup by using $N$ processors

\[ \frac{1}{(1-P) + \frac{P}{N}} \]

\[ \begin{array}{c|c|c}
   & \text{Serial} & \text{Parallel} \\
1 \text{ Processor} & 50 & 50 \Rightarrow 50 \text{ Time}_{\text{new}} \\
2 \text{ Processors} & 50 \text{ Time}_{\text{parallel}} & 25 \text{ Time}_{\text{parallel}} \\
\end{array} \]

\[ \text{Speedup} = \frac{1}{0.5 + 0.5/2} = 1.33 \ldots \]

Amdahl’s law

- Given a proportion $P$ of a program that can be made parallel, and the remaining serial portion $(1-P)$, speedup by using $N$ processors

\[ \frac{1}{(1-P) + \frac{P}{N}} \]

\[ \begin{array}{c|c|c|c}
   & \text{Serial} & \text{Parallel} & \text{Processors} \\
1 \text{ Processor} & 50 & 50 \Rightarrow 50 \text{ Time}_{\text{new}} \\
1 \text{ Processor} & 50 \text{ Time}_{\text{parallel}} & 50 \text{ Time}_{\text{parallel}} \\
\end{array} \]

\[ \text{Speedup} = \frac{1}{0.5 + 0} = 2 \]
**Types of Multiprocessors (MPs)**

- **UMA MP**
  - Uniform Memory Access
  - Access to all memory occurs at the same speed for all processors.

- **NUMA MP**
  - Non-uniform memory access
  - Access to some parts of memory is faster for some processors than other parts of memory

- We will focus on UMA

**Bus Based UMA**

- Simplest MP is more than one processor on a single bus connect to memory (a)
  - Bus bandwidth becomes a bottleneck with more than just a few CPUs

**Cache Consistency**

- What happens if one CPU writes to address 0x1234 (and it is stored in its cache) and another CPU reads from the same address (and gets what is in its cache)?

**Bus Based UMA**

- To further scale the number processors, we give each processor private local memory
  - Keep private data local on off the shared memory bus
  - Bus bandwidth still becomes a bottleneck with many CPUs with shared data
  - Complicate application development
  - We have to partition between private and shared variables
Multi-core Processor

- With only a single shared bus, scalability is limited by the bus bandwidth of the single bus
  - Caching only helps so much
- Alternative bus architectures do exist.

Bus Based UMA

- Pro
  - Any CPU can access any available memory with less blocking
- Con
  - Number of switches required scales with \( n^2 \)
  - \( 1000 \) CPUs need \( 1000000 \) switches

UMA Crossbar Switch

- Pro
  - Any CPU can access any available memory with less blocking
- Con
  - Number of switches required scales with \( n^2 \)
  - \( 1000 \) CPUs need \( 1000000 \) switches

Summary

- Multiprocessors can
  - Increase computation power beyond that available from a single CPU
  - Share resources such as disk and memory
- However
  - Assumes parallelizable workload to be effective
  - Assumes not I/O bound
  - Shared buses (bus bandwidth) limit scalability
    - Can be reduced via hardware design
    - Can be reduced by carefully crafted software behaviour
    - Good cache locality together with limited data sharing where possible

Question

- How do we construct an OS for a multiprocessor?
  - What are some of the issues?
Each CPU has its own OS

- Statically allocate physical memory to each CPU
- Each CPU runs its own independent OS
- Share peripherals
- Each CPU (OS) handles its processes system calls

Master-Slave Multiprocessors

- OS (mostly) runs on a single fixed CPU
  - All OS tables, queues, buffers are present/manipulated on CPU 1
- User-level apps run on the other CPUs
  - And CPU 1 if there is spare CPU time
- All system calls are passed to CPU 1 for processing

Issues

- Each processor has its own scheduling queue
  - We can have one processor overloaded, and the rest idle
- Each processor has its own memory partition
  - We can a one processor thrashing, and the others with free memory
- Consistency is an issue with independent disk buffer caches and potentially shared files

Master-Slave Multiprocessors

- Very little synchronisation required
  - Only one CPU accesses majority of kernel data
- Simple to implement
- Single, centralised scheduler
  - Keeps all processors busy
- Memory can be allocated as needed to all CPUs

Issue

- Master CPU can become the bottleneck
- Cross CPU traffic is slow compare to local
Symmetric Multiprocessors (SMP)

- OS kernel run on all processors
  - Load and resource balance between all processors
  - Including kernel execution
- Issue: Real concurrency in the kernel
  - Need carefully applied synchronisation primitives to avoid disaster
- One alternative: A single mutex that makes the entire kernel a large critical section
  - Only one CPU can be in the kernel at a time
  - Only slightly better solution than master slave
  - Better cache locality
  - The "big lock" becomes a bottleneck when in-kernel processing exceeds what can be done on a single CPU
- Better alternative: identify largely independent parts of the kernel and make each of them their own critical section
  - Allows more parallelism in the kernel
  - Code is mostly similar to uniprocessor code
  - Hard part is identifying independent parts that don’t interfere with each other
  - Example:
    - Given a “big lock” kernel, we divide the kernel into two independent parts with a lock each
    - Some kernel activities require more than one part of the kernel
    - Great opportunity to deadlock!!!
    - Results in potentially complex lock ordering schemes that must be adhered to
  - Example:
    - Given the “big lock” kernel, we divide the kernel into two independent parts with a lock each
    - Good chance that one of these locks will become the next bottleneck
    - Subdivision in practice is (in reality) making more code multithreaded (parallelised)
- Real life Scalability Example
  - Early 1990’s, CSE wanted to run 80 X-Terminals off one or more server machines
  - Winning tender was a 4-CPU bar-fridge-sized machine with 256M of RAM
  - Eventual config 6-CPU and 512M of RAM
  - Machine ran fine in all pre-session testing
Real life Scalability Example

- Students + assignment deadline = machine unusable

Real life Scalability Example

- To fix the problem, the tenderer supplied more CPUs to improve performance (number increased to 8)
  - No change????
- Eventually, machine was replaced with
  - Three 2-CPU pizza-box-sized machines, each with 256M RAM
  - Cheaper overall
  - Performance was dramatically improved!!!!!!
  - Why?

Real life Scalability Example

- Paper:
- The 4-8 CPU machine hit a bottleneck in the single threaded VM code
  - Adding more CPUs simply added them to the wait queue for the VM locks, and made others wait longer
- The 2 CPU machines did not generate that much lock contention and performed proportionally better.

Lesson Learned

- Building scalable multiprocessor kernels is hard
- Lock contention can limit overall system performance

SMP Linux similar evolution

- Linux 2.0 Single kernel big lock (1996)
- Linux 2.2 Big lock with interrupt handling locks
- Linux 2.4 Big lock plus some subsystem locks
- Linux 2.6 most code now outside the big lock, data-based locking, lots of scalability tuning, etc, etc.
- Big lock removed in 2011 in kernel version 2.6.39

Multiprocessor Synchronisation

- Given we need synchronisation, how can we achieve it on a multiprocessor machine?
  - Unlike a uniprocessor, disabling interrupts does not work.
    - It does not prevent other CPUs from running in parallel
  - Need special hardware support
Recall Mutual Exclusion with Test-and-Set

enter_region:
TSL REGISTER, LOCK
CMP REGISTER, #0
JNE enter_region
RET | return to caller, critical region entered

leave_region:
MOVE LOCK, #0
RET | return to caller

Entering and leaving a critical region using the TSL instruction

Test-and-Set

- Hardware guarantees that the instruction executes atomically.
  - Atomically: As an indivisible unit.
  - The instruction can not stop half way through

Test-and-Set on SMP

- It does not work without some extra hardware support

Test-and-Set on SMP

- A solution:
  - Hardware locks the bus (one CPU uses the bus exclusively) during the TSL instruction to prevent memory accesses by any other CPU

Test-and-Set on SMP

- Test-and Set is a busy-wait synchronisation primitive
  - Called a spinlock

- Issue:
  - Lock contention leads to spinning on the lock
    - Spinning on a lock requires bus locking which slows all other CPUs down
      - Independent of whether other CPUs need a lock or not
      - Causes bus contention

Test-and-Set on SMP

- Caching does not help reduce bus contention
  - Either TSL still locks the bus
  - Or TSL requires exclusive access to an entry in the local cache
    - Requires invalidation of same entry in other caches, and loading entry into local cache
    - Many CPUs performing TSL simply bounce a single exclusive entry between all caches using the bus
Reducing Bus Contention

- Read before TSL
  - Spin reading the lock variable waiting for it to change
  - When it does, use TSL to acquire the lock
- Allows lock to be shared read-only in all caches until its released
- No bus traffic until actual release
- No race conditions, as acquisition is still with TSL.

```c
start:
while (lock == 1);
r = TSL(lock)
if (r == 1)
goto start;
```


Compares Simple Spinlocks

- Test and Set
  ```c
  void lock (volatile lock_t *l) {
    while (test_and_set(l)) ;
  }
  ```

- Read before Test and Set
  ```c
  void lock (volatile lock_t *l) {
    while (*l == BUSY || test_and_set(l)) ;
  }
  ```

Benchmark

```c
for i = 1 .. 1,000,000 {
  lock(l)
crit_section()
unlock()
compute()
}
```
Other Hardware Provided SMP Synchronisation Primitives

- Atomic Add/Subtract
  - Can be used to implement counting semaphores
- Exchange
- Compare and Exchange
- Load linked; Store conditionally
  - Two separate instructions:
    - Load value using load linked
    - Modify, and store using store conditionally
  - If value changed by another processor, or an interrupt occurred, then store conditionally failed
  - Can be used to implement all of the above primitives
  - Implemented without bus locking


MCS Locks

- Each CPU enqueues its own private lock variable into a queue and spins on it
  - No contention
- On lock release, the releaser unlocks the next lock in the queue
  - Only have bus contention on actual unlock
  - No starvation (order of lock acquisitions defined by the list)

MCS Lock

- Requires
  - `compare_and_swap()`
  - `exchange()`
  - Also called `fetch_and_store()`

```c
#include <stdatomic.h>

atomic_flag lock = __sync_flag_init(0);

// bool acquire_lock(atomic_flag* lock) {
//     if (__sync_lock_test_and_set(lock, 1)) return false;
//     return true;
// }

// bool release_lock(atomic_flag* lock) {
//     if (!__sync_lock_test_and_set(lock, 0)) return false;
//     return true;
// }

void lock_acquire(atomic_flag* lock) {
    while (!acquire_lock(lock));
}

void lock_release(atomic_flag* lock) {
    if (!release_lock(lock)) {
        // handle error
    }
}
```

PROCEDURE release_lock(L : ‘lock, I : ‘quode)
    IF I.next = nil // no known successor
        IF compare_and_swap (L, I, nil)
            RETURN // compare_and_swap returns true iff it swapped
        Repeat while I.next = nil // spin
        I.next->locked := false
```
Selected Benchmark

- Compared
  - test and test and set
  - Others in paper
    - Anderson’s array based queue
    - test and set with exponential back-off
  - MCS

Confirmed Trade-off

- Queue locks scale well but have higher overhead
- Spin Locks have low overhead but don’t scale well

Spinning versus Switching

- Remember spinning (busy-waiting) on a lock made little sense on a uniprocessor
  - There was no other running process to release the lock
  - Blocking and (eventually) switching to the lock holder is the only option.
- On SMP systems, the decision to spin or block is not as clear.
  - The lock is held by another running CPU and will be freed without necessarily blocking the requestor

Spinlocks expect critical sections to be short
Preemption and Spinlocks

- Critical sections synchronised via spinlocks are expected to be short
  - Avoid other CPUs wasting cycles spinning
- What happens if the spinlock holder is preempted at end of holder’s timeslice
  - Mutual exclusion is still guaranteed
  - Other CPUs will spin until the holder is scheduled again!!!!!!

⇒ Spinlock implementations disable interrupts in addition to acquiring locks to avoid lock-holder preemption